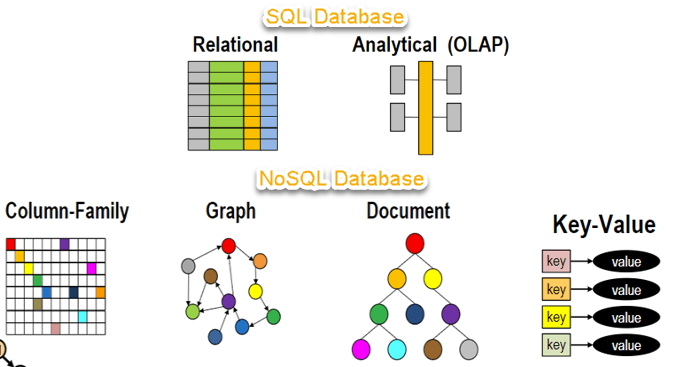
**Ch 1 Concepts of NoSQL: MongoDB**

****

**1.1) NoSQL: Advantage, Disadvantage.**

NoSQL(Non-SQL or Non-Relational): a mechanism for storage and retrieval of data that is modeled in means other than tabular relations used in relational databases.

Such databases have existed since the late 1960s, but the name "NoSQL" was only coined in the early 21st century, triggered by the needs of Web 2.0 companies.

**Benefits:**

1) Develop with agility(agile=iterative software development).

2) Handle data with flexibility.

3) Scalability (scale upon the size of the app)

4) High Performance

5) Availability

6) Highly functional (ideal choice for big data, real-time web apps)

7) Handle large volumes of data at high speed with a scale-out architecture

8) Store unstructured, semi-structured, or structured data

9) Enable easy updates to schemas and fields

10)Be developer-friendly

11)Take full advantage of the cloud to deliver zero downtime.

**Features of NoSQL:**

* Schema flexibility.
* Offers Horizontal scaling.
* NoSQL DB provides quicker queries.
* Avoids joins.
* Easy to use for developers.
* NoSQL databases never follow the relational model.
* Never provide tables with flat fixed-column records.
* Work with self-contained aggregates or BLOBs.
* Doesn’t require object-relational mapping and data normalization
* No complex features like query languages, query planners, referential integrity join, ACID.
* **SCHEMA FREE:**
* NoSQL databases are either schema-free or have relaxed schemas.
* Do not require any sort of definition of the schema of the data.
* Offers heterogeneous structures of data in the same domain.
* **DISTRIBUTED:**
* Multiple NoSQL databases can be executed in a distributed fashion.
* Offers auto-scaling and fail-over capabilities.
* Often ACID concept can be sacrificed for scalability and throughput.
* Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication.

**1.1.1) MongoDB Datatypes:**

**(String,Integer,Boolean,Double,Arrays,Objects)**

**1) String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

**2) Integer** - This type is used to store a numerical value. Integer can be 32-bit or 64-bit depending upon your server.

**3) Boolean** - This type is used to store a boolean (true/ false) value.

**4) Double** - This type is used to store floating-point values.

**5) Arrays** - This type is used to store arrays or lists or multiple values into one key.

**e.g.**

**"skills":[**

**"C",**

**"C++",**

**"JS",**

**"MERN"**

**]**

**6) Objects** -This datatype is used for embedded documents.

**e.g.**

**"book":{**

**"name":"Introduction to Binary",**

**"writer":"Nishant Jadhav"**

**}**

**1.1.2) DB Creation and Dropping DB:**

**Creating DB:**

MongoDB ‘use DATABASE\_NAME’ is used to create database. The command will create a new database if it doesn't exist, otherwise, it will return the existing database.

**Syntax:**

**use DATABASE\_NAME**

**Example:**

**use mydb**

**Currently selected DB:**

**db**

**Check DB list:**

**show dbs**

To see DB in the list, we need to add at least one document to it.

**Dropping DB:**

**dropDatabase()** method is used to drop database.

**Syntax:**

**db.dropDatabase()**

**Switching and dropping existing mydb:**

**use mydb**

**db.dropDatabase**

**1.2) Creating and Drop Collections:**

A collection in MongoDB is a group of documents. MongoDB creates a collection automatically when we insert some document.

**Creating Collection:**

**Syntax:**

**db.createCollection(name, options)**

name: – **String** – name of collection to be created.

option: – **Document** – (optional) specify options about memory size and indexing.

**option includes:**

**capped**(Boolean): fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.

**autoIndexId**(Boolean): if true, automatically create index on \_id fields. Default:false.

**size**(number):maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.

**max**(number)**:** maximum number of documents allowed in the capped collection.

**Example of creating collection:**

**use mydb**

//switched to mydb

**db.createCollection(“firstCollection”)**

//with options:

**db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max : 10000 } )**

//collection is automatically created when document is inserted.

**db.tutorialspoint.insert({"name" : "tutorialspoint"}),**

**WriteResult({ "nInserted" : 1 })**

**Dropping collection:**

**Syntax:**

**db.COLLECTION\_NAME.drop()**

**1.3) CRUD operations (Insert, update, delete, find, Query and Projection operators):**

Create, Read, Update, Delete:

CRUD operations describe the conventions of a user interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it’s standardized according to HTTP action verbs.

**CREATE OPERATION (insert):**

For MongoDB CRUD, if the specified collection doesn’t exist, the create operation will create the collection when it’s executed.

**Two methods:**

1. db.collection\_name.insertOne()

Allows single entry/single document

**db.RecordsDB.insertOne({**

**name: "Marsh",**

**age: "6 years",**

**species: "Dog",**

**ownerAddress: "380 W. Fir Ave",**

**chipped: true**

**})**

If above create operation is successful, the function will return an object where “acknowledged” is “true” and “insertID” is the newly created “ObjectId.”

**db.RecordsDB.insertOne({**

**... name: "Marsh",**

**... age: "6 years",**

**... species: "Dog",**

**... ownerAddress: "380 W. Fir Ave",**

**... chipped: true**

**... })**

**{**

**"acknowledged" : true,**

**"insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")**

**}**

1. db.collection\_name.insertMany()

We can pass multiple items into collection, separating them with commas.

**db.RecordsDB.insertMany([{**

**name: "Marsh",**

**age: "6 years",**

**species: "Dog",**

**ownerAddress: "380 W. Fir Ave",**

**chipped: true},**

**{name: "Kitana",**

**age: "4 years",**

**species: "Cat",**

**ownerAddress: "521 E. Cortland",**

**chipped: true}**

**])**

**READ OPERATION (find):**

The read operations allow you to supply special query filters and criteria that let you specify which documents you want.

1. **find()**

In order to get all the documents from a collection, we can simply use the find() method on our chosen collection. Executing just the find() method with no arguments will return all records currently in the collection.

**db.RecordsDB.find({"species":"Cat"})**

1. **findOne()**

In order to get one document that satisfies the search criteria, we can simply use the findOne() method on our chosen collection.

**db.RecordsDB.find({"age": "8 years"})**

**UPDATE OPERATION:**

Update operations operate on a single collection, and they are atomic at a single document level.

In MongoDB CRUD, there are three different methods of updating documents:

db.collection.updateOne()

db.collection.updateMany()

db.collection.replaceOne()

1. **db.collection.updateOne()**

We can update a currently existing record and change a single document with an update operation. We use the “$set” key and provide the fields we want to update as a value.we use the “$set” key and provide the fields we want to update as a value.

**db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})**

1. **db.collection.updateMany()**

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items.

**db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})**

1. **db.collection.replaceOne()**

The replaceOne() method is used to replace a single document in the specified collection. replaceOne() replaces the entire document, meaning fields in the old document not contained in the new will be lost.

**db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})**

**DELETE OPERATION:**

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document.

MongoDB has two different methods of deleting records from a collection:

db.collection.deleteOne()

db.collection.deleteMany()

1. **db.collection.deleteOne()**

deleteOne() is used to remove a document from a specified collection on the MongoDB server.

**db.RecordsDB.deleteOne({name:"Maki"})**

1. **db.collection.deleteMany()**

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation.

**db.RecordsDB.deleteMany({species:"Dog"})**

**QUERY AND PROJECTION OPERATORS:**

MongoDB query operator includes comparison, logical, element, evaluation, Geospatial, array, bitwise, and comment operators.

**Query operators:**

1. Comparison($eq,$gt,$gte,$in,$lt,$lte,$ne,$nin)

**1)$eq**

Specifies equality condition. The $eq operator matches documents where the value of a field equals the specified value.

**db.inventory.find( { qty: { $eq: 20 } } )**

**db.inventory.find( { "item.name": { $eq: "ab" } } )**

**2)$gt**

Matches values that are greater than a specified value.

**db.inventory.find( { quantity: { $gt: 20 } } )**

**db.inventory.updateOne(**

**{ "carrier.fee": { $gt: 2 } }, { $set: { "price": 9.99 } }**

**)**

**3)$gte**

Matches values that are greater than or equal to a specified value.

**db.inventory.find( { quantity: { $gte: 20 } } )**

**db.inventory.updateMany(**

**{ "carrier.fee": { $gte: 2 } }, { $set: { "price": 9.99 } }**

)

**4)$in**

Matches any of the values specified in an array.

**db.inventory.find( { quantity: { $in: [ 5, 15 ] } }, { \_id: 0 } )**

**db.inventory.updateMany(**

**{ tags: { $in: [ "home", "school" ] } },**

**{ $set: { exclude: false } }**

**)**

**5)$lt**

Matches values that are less than a specified value.

**db.inventory.find( { quantity: { $lt: 20 } } )**

**db.inventory.updateMany( { "carrier.fee": { $lt: 20 } }, { $set: { price: 9.99 } } )**

**6)$lte**

Matches values that are less than or equal to a specified value.

**db.inventory.find( { quantity: { $lte: 20 } } )**

**db.inventory.updateMany(**

**{ "carrier.fee": { $lte: 5 } }, { $set: { price: 9.99 } }**

**)**

**7)$ne**

Matches all values that are not equal to a specified value.

**db.inventory.find( { quantity: { $ne: 20 } } )**

**db.inventory.updateMany( { "carrier.fee": { $ne: 1 } }, { $set: { "price": 9.99 } } )**

**8)$nin**

Matches none of the values specified in an array.

**db.inventory.find( { quantity: { $nin: [ 5, 15 ] } }, { \_id: 0 } )**

**db.inventory.updateMany(**

**{ tags: { $nin: [ "school" ] } },**

**{ $set: { exclude: true } }**

**)**

1. Logical($and,$not,$nor,$or)

**1)$and**

Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.

db.example.find( {

$and: [

{ x: { $ne: 0 } },

{ $expr: { $eq: [ { $divide: [ 1, "$x" ] }, 3 ] } }

]

} )

db.inventory.find( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )

db.inventory.find( {

$and: [

{ $or: [ { qty: { $lt : 10 } }, { qty : { $gt: 50 } } ] },

{ $or: [ { sale: true }, { price : { $lt : 5 } } ] }

]

} )

**2)$not**

Inverts the effect of a query expression and returns documents that do not match the query expression.

db.inventory.find( { price: { $not: { $gt: 1.99 } } } )

**3)$nor**

Joins query clauses with a logical NOR returns all documents that fail to match both clauses.

db.inventory.find( { $nor: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )

The Above query will select all documents in the inventory collection where:

* the price field value does not equal 1.99 and
* the qty field value is not less than 20 and
* the sale field value is not equal to true

The exception in returning documents that do not contain the field in the $nor expression is when the $nor operator is used with the $exists operator.

db.inventory.find( { $nor: [ { price: 1.99 }, { price: { $exists: false } },

{ sale: true }, { sale: { $exists: false } } ] } )

**4)$or**

Joins query clauses with a logical OR returns all documents that match the conditions of either clause. We can nest $or operator.

db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )

The following query, which contains multiple expressions supplied to $or, may produce an error if there is any document where $x is 0:

db.example.find( {

$or: [

{ x: { $eq: 0 } },

{ $expr: { $eq: [ { $divide: [ 1, "$x" ] }, 3 ] } }

]

} )

1. Element($exists,$type)

**1)$exists**

Matches documents that have the specified field.

db.inventory.find( { qty: { $exists: true, $nin: [ 5, 15 ] } } )

The following query specifies the query predicate a: { $exists: true }:

db.records.find( { a: { $exists: true } } )

**2)$type**

Selects documents if a field is of the specified type.

{ field: { $type: [ <BSON type1> , <BSON type2>, ... ] } }

db.addressBook.find( { "zipCode" : { $type : 2 } } );

db.addressBook.find( { "zipCode" : { $type : "string" } } );

1. Evaluation($expr,$jsonSchema,$mod,$regex,$text,$where)

**1)$expr**

Allows use of aggregation expressions within the query language.

$expr can build query expressions that compare fields from the same document in a $match stage.

If the $match stage is part of a $lookup stage, $expr can compare fields using let variables. See Perform Multiple Joins and a Correlated Subquery with $lookup for an example.

The $eq, $lt, $lte, $gt, $gte placed in an $expr operator can use an index on the from collection referenced in a $lookup stage. Limitations:

* Multikey indexes are not used.
* Indexes are not used for comparisons where the operand is an array or the operand type is undefined.
* Indexes are not used for comparisons with more than one field path operand.

db.monthlyBudget.find( { $expr: { $gt: [ "$spent" , "$budget" ] } } )

db.monthlyBudget.find( { $expr: { $gt: [ "$spent" , "$budget" ] } } )

**2)$jsonSchema**

Validate documents against the given JSON Schema.

db.createCollection( <collection>, { validator: { $jsonSchema: <schema> } } )

db.runCommand( { collMod: <collection>, validator:{ $jsonSchema: <schema> } } )

e.g.

db.createCollection("students", {

validator: {

$jsonSchema: {

bsonType: "object",

required: [ "name", "year", "major", "address" ],

properties: {

name: {

bsonType: "string",

description: "must be a string and is required"

},

year: {

bsonType: "int",

minimum: 2017,

maximum: 3017,

description: "must be an integer in [ 2017, 3017 ] and is required"

},

major: {

enum: [ "Math", "English", "Computer Science", "History", null ],

description: "can only be one of the enum values and is required"

},

gpa: {

bsonType: [ "double" ],

description: "must be a double if the field exists"

},

address: {

bsonType: "object",

required: [ "city" ],

properties: {

street: {

bsonType: "string",

description: "must be a string if the field exists"

},

city: {

bsonType: "string",

"description": "must be a string and is required"

}

}

}

}

}

}

} )

**3)$mod**

Performs a modulo operation on the value of a field and selects documents with a specified result.

{ field: { $mod: [ divisor, remainder ] } }

db.inventory.find( { qty: { $mod: [ 4, 0 ] } } )

db.inventory.find( { qty: { $mod: [ 4 ] } } )

**4)$regex(**Regular expressions (shortened as "regex") are special strings representing a pattern to be matched in a search operation.**)**

Selects documents where values match a specified regular expression.

{ <field>: { $regex: /pattern/, $options: '<options>' } }

{ <field>: { $regex: 'pattern', $options: '<options>' } }

{ <field>: { $regex: /pattern/<options> } }

db.inventory.find( { item: { $not: /^p.\*/ } } )

{ name: { $regex: /acme.\*corp/i, $nin: [ 'acmeblahcorp' ] } }

{ name: { $regex: /acme.\*corp/, $options: 'i', $nin: [ 'acmeblahcorp' ] } }

{ name: { $regex: 'acme.\*corp', $options: 'i', $nin: [ 'acmeblahcorp' ] } }

**5)$text**

$text performs a text search on the content of the fields indexed with a text index. A $text expression has the following syntax:

{

$text:

{

$search: <string>,

$language: <string>,

$caseSensitive: <boolean>,

$diacriticSensitive: <boolean>

}

}

$diacriticSensitive:

For diacritic sensitive search, if the suffix stem contains the diacritic mark or marks, the $text operator matches on the exact word.

db.articles.find( { $text: { $search: "coffee" } } )

Query specifies a $search string of three terms delimited by space:

db.articles.find( { $text: { $search: "bake coffee cake" } } )

Exact phrase:

db.articles.find( { $text: { $search: "\"coffee shop\"" } } )

**6)$where**

Use the $where operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system.

db.players.find( { $where: function() {

return (hex\_md5(this.name) == "9b53e667f30cd329dca1ec9e6a83e994")

} } );

1. Geospatial($geoIntersects,$geoWithin,$near,$nearSphere)

GeoJSON object:

{

<location field>: {

$geoIntersects: {

$geometry: {

type: "<GeoJSON object type>" ,

coordinates: [ <coordinates> ]

}

}

}

}

**1)$geoIntersects**

Selects geometries that intersect with a GeoJSON geometry. The 2dsphere index supports $geoIntersects.

**2)$geoWithin**

Selects geometries within a bounding GeoJSON geometry. The 2dsphere and 2d indexes support $geoWithin.

**3)$near**

Returns geospatial objects in proximity to a point. Requires a geospatial index. The 2dsphere and 2d indexes support $near.

**4)$nearSphere**

Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The 2dsphere and 2d indexes support $nearSphere.

1. Array($all,$elemMatch,$size)

**1)$all**

Matches arrays that contain all elements specified in the query.

{ tags: { $all: [ "ssl" , "security" ] } }

db.inventory.find( {

qty: { $all: [

{ "$elemMatch" : { size: "M", num: { $gt: 50} } },

{ "$elemMatch" : { num : 100, color: "green" } }

] }

} )

**2)$elemMatch**

Selects documents if element in the array field matches all the specified $elemMatch conditions.

db.scores.find(

{ results: { $elemMatch: { $gte: 80, $lt: 85 } } }

)

**3)$size**

The $size operator matches any array with the number of elements specified by the argument.

db.collection.find( { field: { $size: 2 } } );

1. Bitwise($bitsAllClear,$bitsAllSet,$bitsAnyClear,$bitsAnySet)

**1)$bitsAllClear**

Matches numeric or binary values in which a set of bit positions all have a value of 0.

**2)$bitsAllSet**

Matches numeric or binary values in which a set of bit positions all have a value of 1.

**3)$bitsAnyClear**

Matches numeric or binary values in which any bit from a set of bit positions has a value of 0.

**4)$bitsAnySet**

Matches numeric or binary values in which any bit from a set of bit positions has a value of 1.

1. Comment

The $comment query operator associates a comment to any expression taking a query predicate.

db.collection.find( { <query>, $comment: <comment> } )

db.records.find(

{

x: { $mod: [ 2, 0 ] },

$comment: "Find even values."

}

)

**1.4 Operators (Projection, update, limit (), sort ()) and Aggregation commands**

1. **Projection**

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The positional $ operator limits the contents of an <array> to return the first element that matches the query condition on the array. Use $ in the projection document of the find() method or the findOne() method when you only need one particular array element in selected documents.

db.mycol.find({},{"title":1,\_id:0})

{"title":"MongoDB Overview"}

{"title":"NoSQL Overview"}

{"title":"Tutorials Point Overview"}

1. **Update**

The following modifiers are available for use in update operations, for example, in db.collection.updateMany() and db.collection.findAndModify().

**Fields:**

**currentDate:** Sets the value of a field to current date, either as a Date or a Timestamp.

**inc:** Increments the value of the field by the specified amount.

**min:** Only updates the field if the specified value is less than the existing field value.

**max:** Only updates the field if the specified value is greater than the existing field value.

**mul:** Multiplies the value of the field by the specified amount.

**rename:** Renames a field

**set:** Sets the value of a field in a document.

**setOnInsert:** Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.

**unset:** Removes the specified field from a document.

1. **limit()**

In MongoDB, the limit() method limits the number of records or documents that you want. It basically defines the max limit of records/documents that you want. Or in other words, this method uses on cursor to specify the maximum number of documents/ records the cursor will return.

db.collection.find(<query>).limit(<number>)

The behavior of limit() is undefined for values less than **-231** and greater than **231**.

1. **sort()**

sort method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

db.COLLECTION\_NAME.find().sort({KEY:1})

Default sort Ascending with sort method. For descending order:

db.mycol.find({},{"title":1,\_id:0}).sort({"title":-1})

1. **Aggregation Command:**

(aggregate, count, distinct, mapReduce)

1. **aggregate**

Performs aggregation tasks such as $group using an aggregation pipeline.

{

aggregate: "<collection>" || 1,

pipeline: [ <stage>, <...> ],

explain: <boolean>,

allowDiskUse: <boolean>,

cursor: <document>,

maxTimeMS: <int>,

bypassDocumentValidation: <boolean>,

readConcern: <document>,

collation: <document>,

hint: <string or document>,

comment: <any>,

writeConcern: <document>,

let: <document> // Added in MongoDB 5.0

}

db.orders.aggregate([

{ $match: { status: "A" } },

{ $group: { \_id: "$cust\_id", total: { $sum: "$amount" } } },

{ $sort: { total: -1 } }

],

{ explain: true }

)

1. **count**

Counts the number of documents in a collection or a view.

{ $count: <string> }

db.collection.aggregate( [

{ $group: { \_id: null, myCount: { $sum: 1 } } },

{ $project: { \_id: 0 } }

] )

1. **distinct**

Displays the distinct values found for a specified key in a collection or a view.

db.runCommand ( { distinct: "inventory", key: "dept" } )

db.runCommand(

{

distinct: "restaurants",

key: "rating",

query: { cuisine: "italian" },

readConcern: { level: "majority" }

}

)

1. **mapReduce**

Performs map-reduce aggregation for large data sets.

db.runCommand( {

mapReduce: <string>,

map: <string or JavaScript>,

reduce: <string or JavaScript>,

finalize: <string or JavaScript>,

out: <output>,

query: <document>,

sort: <document>,

limit: <number>,

scope: <document>,

jsMode: <boolean>,

verbose: <boolean>,

bypassDocumentValidation: <boolean>,

collation: <document>,

writeConcern: <document>,

comment: <any>

} )

[**MongoDB cheat sheet**](https://gist.github.com/bradtraversy/f407d642bdc3b31681bc7e56d95485b6)